

Dependence Analysis of Java Bytecode

Jianjun Zhao

Department of Computer Science and Engineering

Fukuoka Institute of Technology

3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-02, Japan

Email:zhao@cs.fit.ac.jp

Abstract

Understanding program dependencies in a computer program is essential for many software engineering tasks such as program understanding, testing, debugging, reverse engineering, and maintenance. In this paper, we present an approach to dependence analysis of Java bytecode, and discuss some applications of our technique, which include Java bytecode slicing, understanding, and testing.

1 Introduction

Java is a new object-oriented programming language and has achieved widespread acceptance because it emphasizes portability [1]. In Java, programs are being compiled into a portable binary format call *bytecode*. Every class is represented by a single *class file* containing class related data and bytecode instructions (Figure 1 shows a simple Java class *Test* and its corresponding bytecode instructions). These files are loaded dynamically into an interpreter, i.e., the Java Virtual Machine (JVM) [14] and executed. Recently, more and more Java applications are routinely transmitted over the internet as compressed class file archives (i.e., zip files and jar files). A typical example of this situation is to download a web page that contains one or more java applets. However, this situation leads to problems as follows. First, instead of class files, the source code of an application is usually unavailable for the user. If there are some bugs in the classfiles, you need to report the bugs to the application developers and possibly have to pay for a new version of the bug-free software. However, if the developers are not available to support the application (i.e., they do not want to support the software anymore, or they are out of business), the user is the only one that can make changes to fix the bugs. Second, the bytecode instructions contained in a classfile can also have bugs since the methods used for Java software testing do not necessarily remove all possible bugs from its source program. For these cases, if we have some tools that support bytecode understanding, testing, and debugging, we can benefit from them greatly. As a result, the development of techniques and tools to support such kinds of tasks for Java bytecode is important.

One way is to use program dependence analysis technique. Program dependencies are dependence relationships holding between program elements in a program that are implicitly determined by the control flows and data flows in the program. Intuitively, if the computation of a statement directly or indirectly affects the computation of another statement in a program, there might exist some program dependence between the statements. Dependence analysis is the process to determine the program's dependencies by analyzing control flows and data flows in the program.

Many compiler optimizations and program analysis and testing techniques rely on program dependence information, which is topically represented by a *dependence-based representation* (DBR), for example, a *program dependence graph* (PDG) [6, 12]. The PDG, although originally proposed for compiler optimizations, has been used for performing program slicing and for various software engineering tasks such as program debugging, testing, maintenance, and complexity measurements [2, 3, 10, 16, 17]. For example, program slicing, a decomposition technique that extracts program elements related to a particular computation, is greatly benefited from a PDG on which the slicing problem can be reduced to a vertex reachability problem [16] that is much simpler than its original algorithm [18].

Dependence analysis was originally focused on procedural programs. Recently, as object-oriented software become popular, researchers have applied dependence analysis to object-oriented programs to represent various object-oriented features such as classes and objects, class inheritance, polymorphism and dynamic binding [4, 11, 13, 15], and concurrency [19, 20]. (for detailed discussions, see related work section).

However, previous work on dependence analysis has mainly focused on programs written in high-level programming languages, rather than programs in low-level programming languages such as JVM. Although there are several dependence analysis techniques for binary executables on different operating systems and machine architectures [5, 9], the existing dependence analysis techniques can not be applied to Java bytecode straightforwardly due to the specific features of JVM. In order

to perform dependence analysis on Java bytecode, we must extend existing dependence analysis techniques for adapting Java bytecode.

In this paper we propose a dependence analysis technique for Java bytecode. To this end, we first identify and define some types of primary dependencies in a Java bytecode program at the intraprocedural level, and then we discuss some applications of our technique such as Java bytecode slicing, understanding, and testing. In addition to these applications, we believe that the dependence analysis technique presented in this paper can also be used as an underlying base to develop other software engineering tools for Java bytecode to aid debugging, reengineering, and reverse engineering.

The rest of the paper is organized as follows. Section 2 considers the intraprocedural dependence analysis of Java bytecode. Section 3 discusses some applications of the dependence analysis technique. Section 4 discusses some related work. Concluding remarks are given in Section 5.

2 Dependence Analysis

To perform intraprocedural dependence analysis on bytecode methods, it is necessary to identify all primary dependencies in a bytecode method. In this section, we present two types of primary dependencies in a bytecode method.

2.1 Background

We give some definitions that are necessary for formally defining dependencies in a bytecode method from a graphical viewpoint.

A *digraph* is an ordered pair (V, A) , where V is a finite set of elements called *vertices*, and A is a finite set of elements of the Cartesian product $V \times V$, called *arcs*, i.e., $A \subseteq V \times V$ is a binary relation on V . For any arc $(v_1, v_2) \in A$, v_1 is called the *initial vertex* of the arc and said to be *adjacent to* v_2 , and v_2 is called *terminal vertex* of the arc and said to be *adjacent from* v_1 . A *predecessor* of a vertex v is a vertex adjacent to v , and a *successor* of v is a vertex adjacent from v . The *in-degree* of vertex v , denoted by $\text{in-degree}(v)$, is the number of predecessors of v , and the *out-degree* of a vertex v , denoted by $\text{out-degree}(v)$, is the number of successors of v . A *simple digraph* is a digraph (V, A) such that no $(v, v) \in A$ for any $v \in V$.

A *path* in a digraph (V, A) is a sequence of arcs (a_1, a_2, \dots, a_k) such that the terminal vertex of a_i is the initial vertex of a_{i+1} for $1 \leq i \leq k-1$, where $a_i \in A$ ($1 \leq i \leq k$), and k ($k \geq 1$) is called the *length* of the path. If the initial vertex of a_1 is v_I and the terminal vertex of a_k is v_T , then the path is called a path from v_I to v_T .

A *control flow graph* (CFG) of a bytecode method M is a 4-tuple $G_{cfg} = (V, A, s, T)$, where (V, A) is a simple digraph such that V is a set of vertices representing bytecode instructions in M , and $A \subseteq V \times V$ is a set of arcs which represent possible flow of control between vertices in M . $s \in V$ is a unique vertex, called *start vertex* which represents the entry point of M , such that $\text{in-degree}(s) = 0$, and $T \subset V$ is a set of vertices, called *termination vertices* which represent the exit points of M , such that for any $t \in T$ $\text{out-degree}(t) = 0$ and $t \neq s$, and for any $v \in V$ ($v \neq s$ and v does not belong to T), $\exists t \in T$ such that there exists at least one path from s to v and at least one path from v to t .

Traditional control flow analysis represents each statement of a program as a vertex in its CFG. When analyzing Java bytecode, we represent a bytecode instruction as a vertex in the CFG. In our CFG, each vertex represents a bytecode instruction, and each arc represents the possible control of flow between bytecode instructions. Moreover, our CFG contains one unique vertex s to represent the entry point of the method, and a set of termination vertices T to represent the multiple exit points of the method due to the existence of exception handling.

Let u and v be two vertices in the CFG of a bytecode method. u *forward dominates* v iff every path from v to $t \in T$ contains u . u *properly forward dominates* v iff u forward dominates v and $u \neq v$. u *strongly forward dominates* v iff u forward dominates v and there exists an integer k ($k \leq 1$) such that every path from v to $t \in T$ whose length is greater than or equal to k contains u . u is called the *immediate forward dominator* of v iff u is the first vertex that properly forward dominates v in every path from v to $t \in T$.

A *definition-use graph* (DUG) of a bytecode method M is a 4-tuple $G_{dug} = (G_{cfg}, \Sigma, D, U)$, where $G_{cfg} = (V, A, s, T)$ is a CFG of M , Σ is a finite set of symbols, called *local variables* in M , $D : V \rightarrow P(\Sigma)$ and $U : V \rightarrow P(\Sigma)$ are two partial functions from V to the power set of Σ .

The functions D and U map a vertex in G_{cfg} to the set of local variables defined and used, respectively, in the instruction represented by the vertex. A local variable x is defined in an instruction i if an execution of s assigns a value to x , while a variable x is used in an instruction if an execution of s requires the value of x to be evaluated. In Section 2.3, we will show how to determine the definition-use information for bytecode instructions.

Based on the CFG and/or DUG of a bytecode method, we can define control dependence and data dependence in the method.



Figure 1: A simple bytecode method.

2.2 Control Dependencies

Control dependencies represent control conditions on which the execution of an instruction depends on a bytecode method. Informally, an instruction u is directly control-dependent on a control transfer instruction if whether u is executed or not is directly determined by the evaluation result of v .

Let $G_{cfg} = (V, A, s, T)$ be the CFG of a bytecode method, and $u, v \in V$ be two vertices of G_{cfg} . u is *directly strongly control-dependent* on v iff there exists a path $P = (v_1 = v, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n = u)$ from v to u such that P does not contain the immediate forward dominator of v and there exists no vertex v' in P such that the path from v' to u does not contain the immediate forward dominator of v' . u is *directly weakly control-dependent* on v iff v has two successor v' and v'' such that there exists a path $P = (v_1 = v, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n = u)$ from v to u and any vertex v_i ($1 \leq i \leq n$) in P strongly forward dominates v' but does not strongly forward dominate v'' .

Control dependencies represent bytecode instructions related to control conditions on which the execution of an instruction depends. There are three types of JVM instructions that may cause control dependencies.

First, JVM has control transfer instructions that can cause conditionally or unconditionally the JVM to continue execution with an instruction other than the one following the control transfer instructions. Therefore, these kinds of instructions can cause control dependencies.

- Unconditional branch instructions: `goto`, `goto_w`,

`jsr`, `jsr_w`, and `ret`.

- Conditional branch instructions: `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`, `lcmp`, `fcmlpl`, `fcmpg`, `dcmpl`, `dcmpg`.
- Compound conditional branch instructions: `tableswitch` and `lookupswitch`.

Second, in JVM, when the execution of a method is finished, the method must return the control to its caller. The caller is often expecting a value from the called method. JVM provides six return instructions for this purpose, which include `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, and `return`. Since these return instructions can also change the flow of control for the instruction execution, they form another source of control dependencies.

Third, another kind of special branch is the `jsr`, for jump subroutine. It is like a `goto` that remembers where it came from. When `jsr` is executed, it branches to the location specified by the label, and it leaves a special kind of value on the stack called a `returnAddress` to represent the return address. This may cause some control dependence.

Fourth, exceptions are sort of super-`goto` which can transfer control not only within a method, but even terminate the current method to find its destination further up the Java stack. Instructions that may explicitly or implicitly throw an exception can also cause control dependencies because it can explicitly or implicitly change the control flow from one instruction to another. These kind of instructions form another source

of control dependencies.

- Instruction that may explicitly throw an exception: `athrow`.
- Instructions that may implicitly throw an exception: `aaload`, `aastore`, `anewarray`, `arraylength`, `baload`, `bastore`, `caload`, `castore`, `checkcast`, `daload`, `dastore`, `faload`, `fastore`, `getfield`, `getstatic`, `iaload`, `iastore`, `idiv`, `instanceof`, `invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`, `irem`, `laload`, `lastore`, `ldc`, `ldc_w`, `ldc2_w`, `ldiv`, `lrem`, `monitorenter`, `monitorexit`, `multianewarray`, `new`, `newarray`, `putfield`, `putstatic`, `saload`, and `sastore`.

2.3 Data Dependencies

Data dependencies represent the data flow between instructions in a bytecode method. Informally an instruction u is directly data-dependent on another instruction v if the value of a variable computed at v has a direct influence on the value of a variable computed at u .

Let $G_{cfg} = (V, A, s, T)$ be the CFG of a bytecode method, and $u, v \in V$ be two vertices of G_{cfg} . u is *directly data-dependent* on v iff there exists a path $P = (v_1 = v, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n = u)$ from v to u such that $(D(v) \cap U(v) - D(P')) \neq \emptyset$ where $D(P') = D(v_2) \cup \dots \cup D(v_{n-1})$.

We can compute data dependencies by determining the definition and use information, i.e., the set D and U of each instruction first, and compute data dependencies based on such kind of information.

In order to define the data dependencies in a bytecode method, we use an annotated CFG, namely, the DUN, whose vertices are as the same as its CFG, and annotated in two functions according to the definition 2.5. First, there is a function $D(v)$ for the set of all local variables defined at vertex v . Second, there is a function $U(v)$ for the set of all local variables used at vertex v . To construct the DUG of a bytecode method, we should define these two functions explicitly. Intuitively, a use of a local variable corresponds to reading the value of that variable, whereas a definition of a local variable corresponds to writing a value into it.

According to JVM, once a method is invoked, a fixed-sized frame is allocated, which consists of a fixed-sized operand stack and a set of local variables. Effectively this latter set consists of an array of words in which local variables are addressed as word offsets from the array base.

First, we can determine the definition information, i.e., the set D , of each instruction in a bytecode method as follow:

- A bytecode instruction that assigns a value to a local variable in this frame forms a definition of that variable. Therefore, the instructions `istore_<n>`, `istore`, `iinc`, `fstore_<n>`, `fstore`, `astore_<n>`, and `astore` form definitions of the local variable that is defined either implicitly in the opcode or explicitly in the next operand byte (or two bytes if combined with a wide instruction). Similarly, because the instructions `dstore_<n>`, `dstore`, `lstore_<n>`, and `lstore` effectively operate on two local variables (viz. a data item of type long or double at n effectively occupies local variables n and $n+1$), we let each such instruction form two definitions of local variables.

For example, instruction `iinc 5 1` forms a definition of local variable 5, while `dstore_0` forms definition of both local variables 0 and 1.

- The parameter passing mechanism of bytecode causes another source of definitions of local variables: if w words of parameters are passed to a particular method, then invoking that method forms initial definitions of the first w local variables. The types of these parameters can be easily determined from the bytecode context. For an instance method, the first parameter is a reference `this` to an instance of the class in which the method is defined. Types of all other arguments are defined by the corresponding method descriptor.

Second, we can determine the use information, i.e., the set U , of each instruction in a bytecode method as follow:

- A bytecode instruction that reads the value of a local variable forms a use of that variable. Therefore, the instructions `iload_<n>`, `iload`, `iinc`, `fload_<n>`, `fload`, `aload_<n>`, and `aload` forms uses of a single local variable defined either implicitly in the opcode or explicitly in the next operand byte (or two bytes if combined with a wide instruction). Similarly, instructions `dload_<n>`, `dload`, `lload_<n>`, and `lload` effectively form uses of two local variables. At implementation level, each use in a particular method may be represented by two words: the address of the using instruction and the offset of the used local variable.

Once the sets D and U for each instruction of a bytecode method have been determined, the DUG of the method can be constructed. Based on the DUG, it is straightforward to compute data dependencies between instructions in a method.

3 Applications

The intraprocedural dependence analysis technique presented in this paper are useful for many software en-

gineering tasks related to Java bytecode development. Here we briefly describe three tasks: bytecode slicing, understanding, and testing.

3.1 Bytecode Slicing

One of our purpose for analyzing dependencies in a bytecode program is to compute static slices of the program. In this section, we informally define some notions about statically slicing of a bytecode program, and show how to compute static slices of a bytecode program based on dependence analysis.

A *static backward slicing criterion* for a bytecode program is a tuple (s, v) , where s is an instruction in the program and v is a local variable used at s . A *static backward slice* $SS(s, v)$ of a bytecode program on a given static slicing criterion (s, v) consists of all instructions in the program that possibly affect the value of the local variable v at s .

Similarly, we can informally define some notions of forward static slicing of a bytecode program.

A *static forward slicing criterion* for a bytecode program is a tuple (s, v) , where s is an instruction in the program and v is a local variable defined at s . A *static forward slice* $SS(s, v)$ of a bytecode program on a given static slicing criterion (s, v) consists of all instructions in the program that possibly be affected by the value of the variable v at s .

In addition to slicing a complete bytecode program, we can also perform slicing on a single bytecode method independently based on dependence analysis of the method. This may be helpful for locally analyzing a single method.

3.2 Bytecode Understanding

Sometimes it is necessary to understanding a bytecode program. For example, in the case that we can only get the class files of a Java application, but can not get the source code of the application. When we attempt to understand the behavior of a bytecode program, we often want to know which local variables in which bytecode instructions might affect a local variable of interest, and which local variables in which bytecode instructions might be affected by the execution of a variable of interest in the program. As discussed above, the backward and forward slicing of a bytecode program can satisfy the requirements. On the other hand, one of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. When we have to modify a bytecode program, it is necessary to know which local variables in which instructions will be affected by a modified variable, and which local variables in which instructions will affect a modified variable. The needs can be satisfied by backward and forward slicing

the bytecode program being modified.

3.3 Bytecode Testing

A bytecode program can have bugs since the methods used for Java software testing do not necessarily remove all possible bugs from its source program. So it is necessary to propose some testing methods for Java software at the bytecode level. Since our dependence analysis technique analyzes both control and data dependencies which represent either control or data flow properties in a bytecode program, it is a reasonable step to define some dependence-coverage criteria, i.e., test data selection rules based on covering dependencies, for testing Java software at the bytecode level.

4 Related Work

Although dependence analysis has been applied to various programming languages, it is the first time, to our knowledge, to apply dependence analysis to Java bytecode.

Dependence analysis of object-oriented programs has been studied by Larsen and Harrold [13], Malloy *et al.* [15], and Chen *et al.* [4] who considered sequential object-oriented programs, and by Zhao *et al.* [19, 20] who considered concurrent object-oriented programs. However, although these techniques can analyze various types of dependencies of object-oriented programs, they still lack the ability to analyze Java bytecode.

Dependence analysis of binary executables has been studied by Cifuentes and Fraboulet [5] who built a dependence graph for 80286 machine instructions (used by the DOS operating systems) and perform slicing on the graph, and Larus and Schnarr [9] who also use dependence analysis to perform slicing on ELF format binary executables (used by the multiplatform operating system Solaris). However, due to the specific features of JVM, these techniques can not be applied to Java bytecode straightforwardly.

Perhaps, the most similar work with ours is that presented by Hatcliff *et al.* [7], who introduced a dependence analysis technique for multi-threaded programs with JVM concurrency primitives, to support model construction of multi-threaded Java program. In their technique, Java programs are first translated to an intermediate language called *Jimple*, and then they perform dependence analysis on Jimple code. However, since Jimple was originally introduced as the target language for a Java decompiler (which means, to decompile Java class files (bytecode) to Jimple), the file format in Jimple is different from that in JVM. As a result, their technique to perform dependence analysis on Jimple code is different from ours in the sense that we directly perform dependence analysis on JVM bytecode.

5 Concluding Remarks

In this paper we presented a dependence analysis technique to Java bytecode. We first identified and defined various types of primary dependencies in Java bytecode at the intraprocedural level, and then we discussed some applications of our technique in software engineering tasks related to Java bytecode development including Java bytecode slicing, understanding, and testing. In addition to these applications, we believe that the dependence technique presented in this paper can also be used as an underlying base to develop other software engineering tools to aid debugging, reengineering, and reverse engineering for Java bytecode.

Now we are developing a dependence analysis tool to automatically analyze various types of primary dependencies in a bytecode program and construct the dependence graph for the program. We also intend to use the graph as an underlying representation to develop slicer and testing tool for Java bytecode.

Acknowledgements

The author would like to thank the anonymous referees for their valuable suggestions and comments on earlier drafts of the paper.

REFERENCES

- [1] K. Arnold and J. Gosling, "The Java Programming Language," Addison-Wesley, 1996.
- [2] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.
- [3] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.
- [4] J. L. Chen, F. J. Wang, and Y. L. Chen, "Slicing Object-Oriented Programs," *Proceedings of the APSEC'97*, pp.395-404, Hongkong, China, December 1997.
- [5] C. Cifuentes and A. Fraboulet, "Intraprocedural Static Slicing of Binary Executables," *Proc. International Conference on Software Maintenance*, pp.188-195, October 1997.
- [6] J. Ferrante, K.J. Ottenstein, J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.
- [7] J. Hatchliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng, "Formal Study of Slicing for Multithreaded Programs with JVM Concurrency Primitives," *Proc. the Static Analysis Symposium*, September 1999.
- [8] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.
- [9] J. R. Larus and E. Schnarr, "EEL: Machine-independent Executable Editing," *SIGPLAN Conference on Programming Languages, Design and Implementation*, pp.291-300, June 1995.
- [10] B. Korel, "Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol.24, pp.103-108, 1987.
- [11] A. Krishnaswamy, "Program Slicing: An Application of Object-Oriented Program Dependency Graphs," Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.
- [12] D. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler and Optimizations," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp.207-208, 1981.
- [13] L. D. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.
- [14] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1997.
- [15] B. A. Malloy and J. D. McGregor, A. Krishnaswamy, and M. Medikonda, "An Extensible Program Representation for Object-Oriented Software," *ACM Sigplan Notices*, Vol.29, No.12, pp.38-47, 1994.
- [16] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.
- [17] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.
- [18] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.
- [19] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pp.312-320, August 1996, IEEE Computer Society Press.
- [20] J. Zhao, "Slicing Concurrent Java Programs," *Proc. Seventh IEEE International Workshop on Program Comprehension*, pp.126-133, Pittsburgh, PA USA, May 1999.